

Chapter 2: Getting Started with Python

Contents

Chapter 2: Getting Started with Python	1
DRY.....	3
Code Shape.....	3
Visualize and Debug Programs	4
The Story Behind the Name.....	4
Why Python?	4
Interpreted vs Compiled.....	5
Free and Open Source	5
Portable	5
Object-Oriented Programming	6
The Zen of Python.....	6
IDLE	6
Tutorial 2.1 – Hello World at Python Prompt	7
Create a Python Program File.....	7
Tutorial 2.2 – Hello World!.....	8
Tutorial 2.3 – Hello World! Take Two	9
Tutorial 2.4 – Centimeters to Inches.....	10
Requirements	10
Pseudocode	10
How Does the Program Work?	11
Get Input.....	12
Calculate	12
Display Output	12
Tutorial 2.5 - Average.....	13
Requirements.....	13
Comments.....	13
Title Block.....	13
Get Input.....	14

Calculate	14
Display Output	14
Values and Types	14
Literal Constants	15
Variables	16
The Concept of Type	16
Strongly Typed Languages	16
Python is Not Strongly Typed	17
Declaration of Variables	17
Dangerous Curves Ahead!	17
Don't Use the Same Name for Two Variables	17
A More Subtle Danger	17
Assignment Operator	17
Variable Names and Keywords	19
snake_case Variable Names	19
Constants	20
Statements	21
Operators and Operands	21
Expressions	23
Order of Operations	23
Remainder or Modulus Operator	24
Strings	24
Change Case	26
Escape Sequences	26
The Newline (\n) Character	26
Format with F-strings	27
F-strings Data Formats	28
Type	28
Meaning	28
F-strings Alignment	28
Option	28
Meaning	28

F-strings Examples	29
F-strings Field Width.....	30
F-strings Formatting with Commas	32
F-strings Formatting Numbers	32
F-strings Formatting Currency	33
Asking the User for Input	34
Tutorial 2.6 – Savings Account.....	35
Requirements.....	35
Comments.....	35
Choosing Mnemonic (Easy to Remember) Variable Names	36
Debugging.....	38
Practice, Practice, and More Practice – Neural Plasticity.....	39
My Code Isn’t Working	40
Glossary	40
Assignment Submission.....	41

Time required: 60 minutes

DRY

Don’t Repeat Yourself

If you find yourself repeating something (code), there is probably a better solution.

The opposite of **DRY** is **WET**.

WET, can mean either **We Enjoy Typing** or **Wasting Everyone's Time**, depending on who you ask.

Code Shape

Please group program code as follows.

- Get input
- Calculate
- Display

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

The Story Behind the Name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus".

He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Why Python?

The Python programming language in and of itself is not a particularly interesting programming language. To a programmer accustomed to compiled, strongly typed programming languages such as C, C++, and Java, the Python programming language seems to be a little "sloppy", "loosey goosey" and fraught with pitfalls. In the grand scheme of things, Python is an **extremely important** programming language.

The importance of Python derives not from the language itself, but from the hundreds of independent open-source Python libraries that have been developed by others in such areas as image manipulation, networking, plotting and graphics, engineering and scientific programming, web development, gaming, cryptography, database, geographic information systems (GIS), audio, music, embedded devices, robotics, CyberSecurity, presentation, XML processing, etc. In fact, it is hard to come up with a programming application area where someone has not already supplemented the basic Python programming language with an open-source library designed for use in that application area.

Many of those libraries are Python wrappers for compiled C code providing speed and efficiency not normally associated with interpreted languages such as Python. A long list of such library modules is provided at [UsefulModules](#). Many of the links on that page point to other pages that also contain lists of links. An even longer list is provided on the [SciPy Topical Software](#) page.

Interpreted vs Compiled

A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0's and 1's) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You run the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc. This also makes your Python programs much more portable. You can just copy your Python program onto another computer, and it just works!

A compiled program built in C++ or something similar runs faster, as it is designed for the architecture of the device. The big advantage of Python is that it is very portable to hundreds of different devices.

Free and Open Source

Python is an example of a FLOSS (Free/Libre and Open-Source Software). In simple terms, you can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like Kivy to create games for your computer and for iPhone, iPad, and Android.

Object-Oriented Programming

Python supports procedure-oriented programming as well as object-oriented programming. In procedure-oriented languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In object-oriented languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

The Zen of Python

Run the following code to find out about the Python mission. This includes many good program design and coding principles.

At a Python prompt, type the following code. You should see the following The Zen of Python.

```
# Display the Zen of Python
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

IDLE

IDLE is a simple integrated development environment (IDE) that comes with Python. It's a program that allows you to type in your programs and run them.

When you first start IDLE, it starts up in the interactive shell. This is an interactive command interface where you can type in Python code and see the output in the same window. This is shown in the example below.



```
IDLE Shell 3.9.1
File Edit Shell Debug Options Window Help
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, D64) on win32
Type "help", "copyright", "credits" or "license()"
>>> print ("Hello World!")
Hello World!
>>>
```

Tutorial 2.1 – Hello World at Python Prompt

Time to write and run your first Python program.

1. Start **IDLE**.
2. Type the follow code at the Python prompt.
3. Press **Enter** to run the code.

```
print("Hello World! ")
```

4. You should see Hello World! printed at the Python prompt as shown in the above example.
5. Congratulations! You coded and executed your first successful program in Python!

Create a Python Program File

The Python prompt is great for testing out code and trying things out. A Python program file is where we will be saving and running our Python programs.

1. In **Visual Studio Code** → **File** → **New File** to create a new Python program file.
2. To save a Python program file, go to **File** → **Save As**, give the file a descriptive name, browse to the folder you want to save it in. Click **Save**.

If you right-click on a Python program file, there should be an option called **Edit with Code**.

Python program files end with **.py**.

Keyboard shortcuts: The following keystrokes work in IDLE and can really speed up your work. Most of these shortcuts work with other programs as well.

The most important keyboard shortcut is CTRL+S which saves the file. Get used to doing that on a very regular basis

Keystroke	Result
CTRL+S	Save the file
CTRL+C	Copy selected text
CTRL+X	Cut selected text
CTRL+V	Paste
CTRL+Z	Undo the last keystroke or group of keystrokes
CTRL+SHIFT+Z	Redo the last keystroke or group of keystrokes
F5	Run module

In the following tutorials you will type in and run some sample programs. You are not expected to understand everything yet. The purpose of these tutorials is to give you a taste of what a Python program looks like.

Tutorial 2.2 – Hello World!

This is traditionally the first program anyone writes in a new programming language. It is also a good way to test your development environment to make sure everything is working.

We will write and run the traditional “Hello World!” program from a Python program file.

1. Launch **Visual Studio Code**.
2. Click **File** → **Save As**. Save your program as **hello_world.py**
3. Enter the following code required to print a **Hello World!** message.


```

1  """
2      Name: hello_world.py
3      Author:
4      Created:
5      Purpose: My first Python program
6  """
7
8  # Print the literal string Hello World!
9  print("Hello World!")

```

4. Press **F5** to run the program.
5. Your program should print **Hello World!** in the terminal.
6. Example run:

```

Python 3.9.1 (tags/v3.9.1:1e5d3
(AMD64)] on win32
Type "help", "copyright", "cred
>>>
= RESTART: Z:\_WNCC\Python\Ass
Hello World.py
Hello World!
>>> |

```

Congratulations! You created your first program in Python.

Tutorial 2.3 – Hello World! Take Two

Variables are a key component to programming in any language. A variable reserves primary working memory space for storing and retrieving data.

1. Open your Hello World program for editing.
2. Assign 2 different Hello World messages to 2 different variables. You can create whatever messages you would like.
3. Print out the variables.

Code:

```

1  """
2      Name: hello_world.py
3      Author:
4      Created:
5      Purpose: My first Python program
6  """
7
8  # Print the literal string Hello World!
9  print("Hello World!")
10
11 # Create two string variables and assign values
12 message1 = "Hello Python World! Here I come!"
13 message2 = "Monty Python Rules!!!! ;'"
14
15 # Print the values stored in the variables
16 print(message1)
17 print(message2)

```

Example run:

```

Python 3.9.1 (tags/v3.9.1:1e5d33e
(AMD64)] on win32
Type "help", "copyright", "credit
>>>
= RESTART: Z:\_WNCC\Python\Assign
Hello World2.py
Hello World!
Hello Python World! Here I come!
Monty Python Rules!!!! ;')
>>>

```

Tutorial 2.4 – Centimeters to Inches

Requirements

Create a program that takes Centimeters input from the user and converts this to Inches.

Pseudocode

Pseudocode is a way of planning out your program in simple language. Every program you write solves some type of program. An algorithm is the method of solving that problem. Pseudocode is the first step of translating the algorithm to the programming language. The more you figure out before you code, the easier it is to code an efficient and elegant program. An elegant, simple program is like a work of art.

Prompt the user for Centimeters input

Get input into a variable

Convert Centimeters to Inches

Display the results

1. Create a new Python program.
2. Save the file as **cm_to_inches.py**.
3. Type in the following program.

```
1 """
2     Name: cm_to_inches.py
3     Author:
4     Created:
5     Purpose: Convert Centimeters to Inches
6 """
7
8 # Get centimeters from user
9 print("Centimeter to Inches Converter")
10 centimeters = float(input('Enter centimeters: '))
11
12 # Convert centimeters to inches
13 inches = centimeters * .393701
14
15 # Display inches and centimeters
16 print(f"{centimeters} cm is equal to {inches:.2f} inches.")
```

4. Run the program.
5. The program will run in the shell window. The program will ask you for a temperature. Type in 20 and press enter. The program's output looks something like this:

Example run:

```
Centimeter to Inches Converter
Enter centimeters: 20
20.0 cm is equal to 7.87 inches.
```

How Does the Program Work?

```
1 """
2     Name: cm_to_inches.py
3     Author:
4     Created:
5     Purpose: Convert Centimeters to Inches
6 """
```

This is a multiline comment that gives information about the program. It is typical to have a header to a program.

Get Input

```
8 # Get centimeters from user
9 print("Centimeter to Inches Converter")
10 centimeters = float(input('Enter centimeters: '))
```

The first line is a single comment to let another programmer know what is going on.

The next line announces the program title.

The next line is an input function. It's job is to ask the user to type something in and to capture what the user types. The part in quotes is the prompt that the user sees. It is called a string and it will appear to the program's user exactly as it appears in the code itself.

The float function is one of the ways to get numerical input in Python. Float converts the input string to a floating point (Decimal) number like (PI 3.142).

We use = to assign the float value to `centimeters`.

This is an example of an assignment statement in pseudocode.

```
variable = users's input
```

Calculate

```
12 # Convert centimeters to inches
13 inches = centimeters / 2.54
```

This line uses the standard conversion math formula to convert centimeters to inches. Notice that the = is different in programming than in math. It is called the assignment statement. The calculations or values on the right are assigned to the variable on the left.

Display Output

```
15 # Display inches and centimeters
16 print(f"{centimeters} cm is equal to {inches:.2f} inches.")
```

The last line uses the print function to print out the conversion. The part in quotes is another string and will appear to your program's user exactly as it appears in quotes here. The second argument to the print function is the variable containing the calculations result from the previous step.

This program may seem too short and simple to be of much use. There are many websites that have little utilities that do similar conversions. Their code is not much more complicated than the code here.

Tutorial 2.5 - Average

Requirements

This program computes the average of two numbers that the user enters.

Create a new Python program and save it as **average.py**.

```
1 """
2     Filename: average.py
3     Author:
4     Date:
5     Purpose: Average two numbers input by user
6 """
7
8 # Get input from user
9 number1 = float(input('Enter the first number: '))
10 number2 = float(input('Enter the second number: '))
11
12 # Calculate average
13 average = (number1 + number2) / 2
14
15 # Display average
16 print(f"The average of the numbers you entered is {average}")
```

Comments

Notice the lines that start with #. These are comments to help someone else understand your code. It will also help you remember what you were thinking when you wrote the code. Commenting your code is a good programming practice.

Title Block

```
1 """
2     Filename: average.py
3     Author:
4     Date:
5     Purpose: Average two numbers input by user
6 """
```

The Title Block in a program is part of documenting or commenting a program to make it more understandable. Please use this in each of your programs.

Get Input

```
8 # Get input from user
9 number1 = float(input('Enter the first number: '))
10 number2 = float(input('Enter the second number: '))
```

We need to get two numbers from the user. We get the numbers one at a time and assign each number to a variable.

Notice that the program used ' instead of ". They are interchangeable in Python. They must be in pairs enclosing the string.

Calculate

```
# Calculate average
average = (number1 + number2) / 2
```

Note the parentheses in the average calculation. This is the order of operations. All multiplication and division are performed before any addition and subtraction. We use parentheses to get Python to do the addition first.

Display Output

```
15 # Display average
16 print(f"The average of the numbers you entered is {average}")
```

The output is displayed to the user.

Example run:

```
Enter the first number: 2
Enter the second number: 5
The average of the numbers you entered is 3.5
```

Values and Types

A value is one of the basic things a program works with, like a letter or a number. The types of values we have seen so far are 1, 2, and "Hello, World!"

These values belong to different types: 2 is an integer, and "Hello, World!" is a string literal, so called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

```
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Strings belong to the type str and integers belong to the type int. Numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating point.

```
>>> type(3.2)
<class 'float'>
```

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

They are strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers, which it prints with spaces between.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

Literal Constants

An example of a literal constant is a number like **5**, **1.23**, or a string like **'This is a string'** or **"It's a string!"**.

It is called a literal because it is literal - you use its value literally. The number 2 always represents itself and nothing else - it is a constant because its value cannot be changed. These are referred to as literal constants.

Using literal constants for numbers is called using "magic numbers". There is no way to easily tell what that number means. The number 2 could be 2 cats, 2 percent, etc. It is not a good practice to use literal constants for numbers.

Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value stored in a memory location.

You can also think of a variable as a pigeonhole in memory, which has a nickname, where you can store values.

You can later retrieve the values that you have stored there by referring to the pigeonhole by its nickname (identifier). You can also store a different value in the pigeonhole later if you desire.

Numbers are mainly two types – integers and floats.

- An integer is a whole number, **2**
- A float is a floating-point number, **2.23**
- A string is a sequence of characters, just a bunch of words.

The Concept of Type

Strongly Typed Languages

One of the main differences between Python and programming languages such as Java and C++ is the concept of type.

In strongly typed languages like Java and C++, variables not only have a name, they also have a type. The type determines the kind of data that you can store in the pigeonhole.

It is probably more correct to say that the type determines the values that you can store there and the operations (addition, subtraction, etc.) that you can perform on those values.

Python is Not Strongly Typed

One of the characteristics that makes Python easier to use than Java is the fact that with Python you don't have to be concerned about the type of a variable. Python takes care of type issues for you behind the scenes. That ease of use comes with some costs attached.

Declaration of Variables

Another difference between Python and Java is that with Java, you must declare variables before you can use them. Declaration of variables is not required with Python.

With Python, if you need a variable, you simply come up with a name and start using it as a variable.

Dangerous Curves Ahead!

With this convenience comes some danger. You can only have one variable with the same name within the same scope (More on scope later).

Don't Use the Same Name for Two Variables

With Python, if you unintentionally use the same name for two or more variables, the first will be overwritten by the second. This can lead to program bugs that are difficult to find and fix.

A More Subtle Danger

A more subtle danger is that you create a variable that you intend to use more than once and you spell it incorrectly in one of those uses. This can be an extremely difficult problem to find and fix.

Assignment Operator

The equal sign (=) is the assignment operator in Python. An assignment statement can create new variables and give them values. After the assignment statement, the variable on the left contains the value on the right side of the operator.

```
message = 'And now for something completely different'
number = 17
pi = 3.1415926535897931
```

```

1 message = 'And now for something completely different'
2 number = 17
3 pi = 3.1415926535897931
→ 4 fred = number + pi

```

[Edit this code](#)

→ line that just executed
→ next line to execute

Global frame

message	"And now for something completely different"
number	17
pi	3.1416
fred	20.1416

Python 3.6
(known limitations)

```

1 message = 'And now for something completely different'
2 number = 17
3 pi = 3.1415926535897931
4 fred = number + pi

```

[Edit this code](#)

ne that just executed
ext line to execute

Frames Objects

Global frame

- message → str "And now for something completely different"
- number → int 17
- pi → float 3.1416
- fred → float 20.1416

The diagram above was created by [Python Tutor](#). It shows a couple different views of the sample code. You can paste in code from Python and Java. It shows you what is happening in real time. We will use this throughout this class to see what is happening “inside” our programs.

*"[Online Python Tutor](#) is a free educational tool created by [Philip Guo](#) that helps students overcome a fundamental barrier to learning programming: understanding what happens as the computer executes each line of a program's source code. Using this tool, a teacher or student can write **Python**, **Java**, and **JavaScript** programs in the Web browser and visualize what the computer is doing step-by-step as it executes those programs."*

This example makes three assignments to three different pigeonholes. The first assigns a string to a new variable named message; the second assigns the integer 17 to number; the third assigns the (approximate) value of pi to pi.

To display the value of a variable, you can use a print statement:

```

>>> print(number)
17
>>> print(pi)
3.141592653589793

```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>

>>> type(n)
<class 'int'>

>>> type(pi)
<class 'float'>
```

Variable Names and Keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for. Although you can make up your own names for variables, you must follow these rules:

- You cannot use one of Python's keywords as a variable name.
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct (case sensitive). This means the variable name Number is not the same as number.

Variable names can be long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

snake_case Variable Names

Variables are typically combinations of more than one word. A common convention among programmers in Python is to use snake_case.

- The variable name is all lowercase letters.
- An underscore is placed between each word.

Here are some examples of snake_case.

```
gross_pay
hours_worked
total_hours_worked
```

Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade' SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy' SyntaxError: invalid syntax
```

76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @. But what's wrong with class?

It turns out that class is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 35 keywords:

and	del	From	None	True
as	elif	Global	Nonlocal	Try
assert	else	If	Not	While
break	except	Import	Or	With
class	False	In	Pass	Yield
continue	finally	Is	Raise	Async
def	for	Lambda	Return	Await

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later. You can then use the name of the fixed data value in expressions or programs.

You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

Constants are not enforced in Python. It is what is called a convention. This means that programmers agree to label and use constants in Python.

Constants are used to replace “magic numbers”. A magic number is an unexplained value that appears in a program’s code. $2 + 2 = 4$ are all magic numbers as they have no meaning.

Constants may be used in many parts of the program. Instead of having to search through the program to replace a magic number, you change the constant once.

Constants use the following naming convention. All characters are capitalized with an underscore (_) in between.

```
A_SIMPLE_CONSTANT
ANOTHER_CONSTANT
SALES_TAX
```

Statements

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

Operators and Operands

Symbol	Operation	Example	Description
--------	-----------	---------	-------------

+	Addition	$a + b$	Adds two numbers
-	Subtraction	$a - b$	Subtracts one number from another
-	Negation	$-a$	Change sign of operand
*	Multiplication	$a * b$	Multiplies one number by another
/	Division	a / b	Divides one number by another and gives the result as a floating-point number
//	Integer division	$a // b$	Divides one number by another and gives the result as a whole number
%	Remainder or Modulus	$a \% b$	Divides one number by another and gives the remainder
**	Exponent	$a ** b$	Raises a number to a power

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

The operators `+`, `-`, `*`, `/`, and `**` perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20 + 32
hour - 1
hour * 60 + minute
minute / 60
5**2 (5 + 9) * (15 - 7)
```

The result of this division is a floating-point result:

```
minute = 59
minute / 60

0.9833333333333333
```

In this example, the value is truncated, the fractional part is left out.

```
5 // 2

2
```

Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable `x` has been assigned a value):

```
17 * x
x + 17
```

If you type an expression in interactive mode, the interpreter evaluates it and displays the result:

```
1 + 1

2
```

In a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Type the following statements in the Python interpreter to see what they do:

```
5
x = 5
x + 1
```

Order of Operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

- **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3 - 1)` is 4, and `(1 + 1) ** (5 - 2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.
- **E**xponentiation has the next highest precedence, so `2 ** 1 + 1` is 3, not 4, and `3 * 1 ** 3` is 3, not 27.
- **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. `2 * 3 - 1` is 5, not 4, and `6 + 4 / 2` is 8, not 5.
- **O**perators with the same precedence are evaluated from left to right. The expression `5 - 3 - 1` is 1, not 3, because the `5 - 3` happens first and then 1 is subtracted from 2.

BIG NOTE: When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

Remainder or Modulus Operator

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
quotient = 7 // 3
print(quotient)

2

remainder = 7 % 3
print(remainder)

1
```

7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y`.

You can also extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly, `x % 00` yields the last two digits.

Strings

A variable that holds a sequence of characters text is called a string. Here is an example of strings and string literals.

```
a_string = 'Python is Grand'
```

'Python is Grand!' is called a string literal. String literals must be enclosed by either a pair of single quotes `' '` or a pair of double quotes `" "`.

str is a variable type that stores a string. In the statement above, the string literal is assigned to the string variable `a_string`.

If you want to print a single quote in the string literal, use double quotes to surround the string literal. You would do the opposite if you wanted to use double quotes in the string literal.


```
print("I'm in love with Python!")
```

Output

```
I'm in love with Python!
```

The + operator works with strings, but it is not addition in the mathematical sense. It performs concatenation, which means joining the strings by linking them end to end. For example:

```
first = 10
second = 15
print(first + second)
```

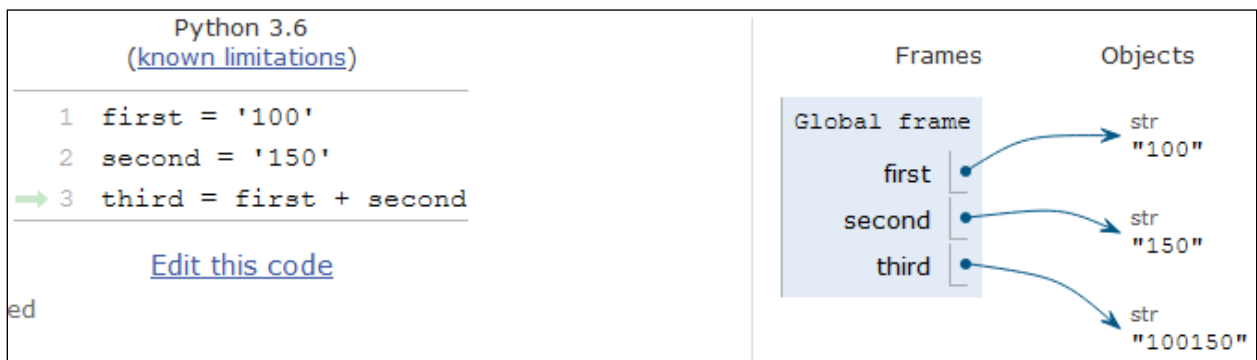
```
25
```

```
first = '100'
second = '150'
print(first + second)
```

```
100150
```

```
# Notice the space after very
print('very ' + 'hot')
```

```
very hot
```



This example shows what happens when strings are concatenated.

The variable named **third** points to a third object of type **str** containing a string that was produced by using the "+" operator to concatenate the contents of two existing objects of type **str**. It is important to note that the contents of the third object contains the concatenation of copies of the contents of the first two objects. It doesn't simply contain pointers to the other two objects.

The `*` operator also works with strings by multiplying the content of a string by an integer. For example:

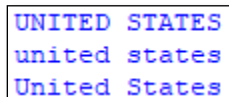
```
first = "Test "  
second = 3  
print(first * second)  
  
Test Test Test
```

Change Case

There are 3 string operators that allow Python to change the case of a string. They can be used to change the case and assign to the same or another string. They can be used in a print statement which doesn't change the original string.

```
message = "United States"  
message = message.upper()  
print(message)  
message = message.lower()  
print(message)  
print(message.title())
```

Example run:



```
UNITED STATES  
united states  
United States
```

Escape Sequences

Escape sequences are special sequences of characters used to represent other characters that:

- cannot be entered directly into a string
- would cause a problem if entered directly into a string

The Newline (`\n`) Character

This is an "escape character" representation of the *newline* character. It appeared in the output at the point representing the end of the first line of input. This indicates that the interpreter knows and remembers that the input string was split across two lines.

As the name implies, a newline character is a character that means, "Go to the beginning of the next line."

The newline character is sort of like the wind. You can't see the wind, but you can see the result of the wind blowing through a tree.

Similarly, you can't normally see a newline character, but you can see what it does. Therefore, we must represent it by something else, like `\n` if we want to be able to see where it appears within a string.

An example of a character that cannot be entered directly into a string is the newline character. Except when using triple quoted strings, you cannot enter the newline character directly into a string.

Why? Because when you press the Enter key in an attempt to enter a newline, that simply terminates your input for that line. It doesn't enter the newline character into the string.

```
print("Dick\nBaldwin")
```

```
Dick
```

```
Baldwin
```

We entered the newline escape sequence between my first and last names when we constructed the string. When the string was printed, the cursor advanced to a new line following my first name and printed my last name on the new line. That is what escape sequences are all about.

Format with F-strings

F-strings, or formatted string literals, are a fairly simple way to format values using the **print** function. They are called **f-strings** because you need to prefix a string with the letter 'f' to get an f-string. The letter 'f' also indicates that these strings are used for formatting.

To use formatted string literals, begin a string with f or F before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between curly braces `{ }` characters that can refer to variables or literal values. **f-strings** support extensive modifiers that control the final appearance of the output string. Expressions in **f-strings** can be modified by a format specification.

F-strings Data Formats

There are many ways to represent strings and numbers when using f-strings. The following table shows the most commonly used.

Type	Meaning
s	String format—this is the default type for strings.
d	Decimal Integer. Outputs the number in base 10.
n	Number. This is the same as d except that it uses the current locale setting to insert the appropriate number separator characters.
e	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.
f	Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6.
n	Number. This is the same as g , except that it uses the current locale setting to insert the appropriate number separator characters.
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

F-strings Alignment

There are several ways to align variables in **f-strings**.

Option	Meaning
<	Forces the field to be left-aligned within the available space (this is the default for most objects).
>	Forces the field to be right aligned within the available space (this is the default for numbers).
=	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default when '0' immediately precedes the field width.

^	Forces the field to be centered within the available space.
---	---

F-strings Examples

The following is a basic example of the use of f-strings:

```
x = 4.5
print(f'This will print out the variable x: {x}')
```

Example run:

```
This will print out the variable x: 4.5
```

The variable, *x*, is enclosed in curly braces (`{ }`) and the f-string understands that *x* is a float and displays it as assigned.

If you want to change the number of decimals displayed, you would write this:

```
x = 4.5
print(f'This will print out the variable x: {x:.3f}')
```

Example run:

```
This will print out the variable x: 4.500
```

A colon is now added after the variable, which now controls the formatting of the float, specifying that three decimal places are to be displayed.

```
x = 4.5
y = 5.6
z = 5000
print(f'This will print out the variable x: {x:.3f} do you see?')
print(f'This prints out y as currency: ${y:.2f} More money for me.')
print(f'This prints out z with commas: {z:,.2f}')
print(f'This prints out z as currency with commas: ${z:,.2f}')
```

Example run:

```
This will print out the variable x: 4.500 do you see?
This prints out y as currency: $5.60 More money for me.
This prints out z with commas: 5,000.00
This prints out z as currency with commas: $5,000.00
```

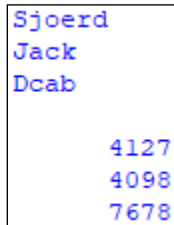
This example combines more than one variable with text. The \$ formats the variable as currency, the , displays with commas.

F-strings Field Width

Passing an integer after the ":" will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
table = ['Sjoerd', 'Jack', 'Dcab']
for name in table:
    print(f'{name:10}')
print()
table2 = [4127, 4098, 7678]
for num in table2:
    print(f'{num:10}')
```

Example run:



```
Sjoerd
Jack
Dcab

      4127
      4098
      7678
```

For the list, table, the strings are left-aligned in a field width of 10, while the numbers in the list, table2, are right-aligned in a field width of 10.

The following lines produce a tidily aligned set of columns giving integers and their squares and cubes. The tab character (\t) can also be used in an f-string to line up columns, particularly when column headings are used:

```
print(f'Number\tSquare\tCube')
for x in range(1, 11):
    print(f'{x:2d}\t\t{x*x:3d}\t\t\t{x*x*x:4d}')
```

Example run:

Number	Square	Cube
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

The following takes the previous program and converts `x` to a `float()`. This demonstrates formatting of floating-point numbers. This also demonstrates how the use of a value for width will enable the columns to line up. The number after the colon `:` sets the width.

```
print(f'Number\tSquare\t      Cube')
for x in range(1, 11):
    x = float(x)
    print(f'{x:5.2f}\t{x*x:6.2f}\t{x*x*x:8.2f}')
```

Example run:

Number	Square	Cube
1.00	1.00	1.00
2.00	4.00	8.00
3.00	9.00	27.00
4.00	16.00	64.00
5.00	25.00	125.00
6.00	36.00	216.00
7.00	49.00	343.00
8.00	64.00	512.00
9.00	81.00	729.00
10.00	100.00	1000.00

The following program demonstrates the use of strings, decimals, and floats, as well as tabs for a type of report that is often produced in a typical Python program. Notice the use of the dollar sign (\$) just before the variables that are to be displayed as prices.

```

APPLES = .50
BREAD = 1.50
CHEESE = 2.25
num_apples = 3
num_bread = 4
num_cheese = 2
prc_apples = 3 * APPLES
prc_bread = 4 * BREAD
prc_cheese = 2 * CHEESE
str_apples = 'Apples'
str_bread = 'Bread'
str_cheese = 'Cheese'
total = prc_apples + prc_bread + prc_cheese
# Center title over 30 spaces
print(f'{"My Grocery List":^30s}')
# Print 30 equals signs =
print(f'{"="*30}')
# Print a tab aligned grocery list
print(f'{str_apples}\t{num_apples:10d}\t${prc_apples:>5.2f}')
print(f'{str_bread}\t{num_bread:10d}\t${prc_bread:>5.2f}')
print(f'{str_cheese}\t{num_cheese:10d}\t${prc_cheese:>5.2f}')
print(f'{"Total:":>19s}\t${total:>4.2f}')

```

F-strings Formatting with Commas

Finally, commas are often need when formatting large numbers. The following shows the use of commas when numbers are aligned and when numbers do not required alignment.

```

number = 1000000
print (f'The number, 1000000, formatted with a comma {number:,.2f}')
print(f'The number, 1000000, formatted with a comma and right-aligned in a
width of 15 {number:>15,.2f}')

```

Example run:

```

The number, 1000000, formatted with a comma 1,000,000.00
The number, 1000000, formatted with a comma and right-aligned in a width of 15      1,000,000.00

```

F-strings Formatting Numbers

Notice the syntax.

- f precedes what we want to print.

- Everything is enclosed with double quotes `""`. You can also use single quotes `''`.
- Variables are enclosed by curly braces `{}`.

```
name = "Eric"
age = 74
# Using F-Strings formatting
print(f"Hello, {name}. You are {age} years old.")
```

Example run:

```
Hello, Eric. You are 74 years old.
```

F-strings Formatting Currency

The following example will print out the second line of the program output to two decimal places with comma separators for thousands.

Each of these specifiers can be used separately in any combination.

1. The dollar sign `$` specifies currency format. `$1000`
2. The comma `,` specifies comma formatting. `1,000`
3. The `.2f` rounds the value to 2 decimal places. `1000.02`
4. Put all three together: `$1,000.02`

```
SALES_TAX = .07
sale = 23.44
total_sale = SALES_TAX * sale \n creates a new line
# Without formatting
print("\nTotal Sale without formatting.")
print(total_sale)
# Using F-Strings formatting
print("\nTotal Sale with F-strings")
print(f'Total Sale: ${total_sale:,.2f}')
```

Example run:

```
Total Sale without formatting.
1.6408000000000003

Total Sale with F-strings
Total Sale: $1.64
```

Asking the User for Input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called `input` that gets input from the keyboard.

When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string.

```
>>> inp = input()
Some silly stuff
>>> print(inp)
Some silly stuff
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to `input` to be displayed to the user before pausing for input:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

The sequence `\n` at the end of the prompt represents a newline, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to `int` using the `int()` function:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
>>> int(speed)
17
>>> int(speed) + 5
22
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

We will see how to handle this kind of error later.

Tutorial 2.6 – Savings Account

Requirements

Create a program that calculates a balance using a constant.

1. Save the program as **savings_account.py**.

```
1  """
2      Name: savings_sccount.py
3      Author:
4      Date:
5      Purpose: Demonstrate the use of constants
6  """
7
8  # Declare interest rate constant
9  INTEREST_RATE = .03
10
11 # Get floating point input from user
12 balance = float(input('Enter your current balance: '))
13
14 # Calculate new balance
15 new_balance = (balance * INTEREST_RATE) + balance
16
17 # Display your current interest
18 # Start by echoing user input
19 print(f'Your current balance is: ${balance:,.2f}')
20 print(f'Your current interest rate is: {INTEREST_RATE * 100}%')
21 print(f'Your new account balance with interest is: ${new_balance:,.2f}')
```

Example run:

```
Enter your current balance: 248
Your current balance is: $248.00
Your current interest rate is: 3.0%
Your new account balance with interest is: $255.44
```

Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and in Python they start with the # symbol:

```
# Compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60      # Percentage of an hour
```

Everything from the # to the end of the line is ignored; it has no effect on the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is much more useful to explain why.

This comment is redundant with the code and useless:

```
v = 5      # Assign 5 to v
```

This comment contains useful information that is not in the code:

```
v = 5      # Velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

Choosing Mnemonic (Easy to Remember) Variable Names

Variable names can contain letters, numbers, and the underscore.

- Variable names cannot contain spaces.
- Variable names cannot start with a number.
- Case matters—for instance, temp and Temp and TEMP are different.

If you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

The Python interpreter sees all three of these programs as the same, but humans see and understand these programs quite differently. Humans will most quickly understand the intent of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names “mnemonic variable names”. The word mnemonic means “memory aid”. We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer’s ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 33 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
for word in words:
    print(word)
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code must be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
for slice in pizza:
    print(slice)
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

If our program is truly about reading data and looking for words in the data, pizza and slice are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program.

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you:

The parts of the code that are defined by Python (for, in, print, and :) are in bold and the programmer-chosen variables (word and words) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

Debugging

At this point, the syntax error you are most likely to make is an illegal variable name, like class and yield, which are keywords, or odd~job and US\$, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
    File "<stdin>", line 1
        month = 09
            ^
SyntaxError: invalid token
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

The runtime error you are most likely to make is a “use before def;” that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so LaTeX is not the same as latex.

At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate $1/2$, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get $/2$, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don’t get an error message; you just get the wrong answer.

Practice, Practice, and More Practice – Neural Plasticity

We are going to provide you with a lot of information, but you will also need to put your brain in gear and write a lot of programs to retain the information. Very few people become expert musicians without a lot of practice. Similarly, very few people become expert programmers without a lot of practice.

A little bit of programming each day will strengthen the neural connections in your brain. Consistent daily practice literally re wires your brain. This is called neural plasticity.

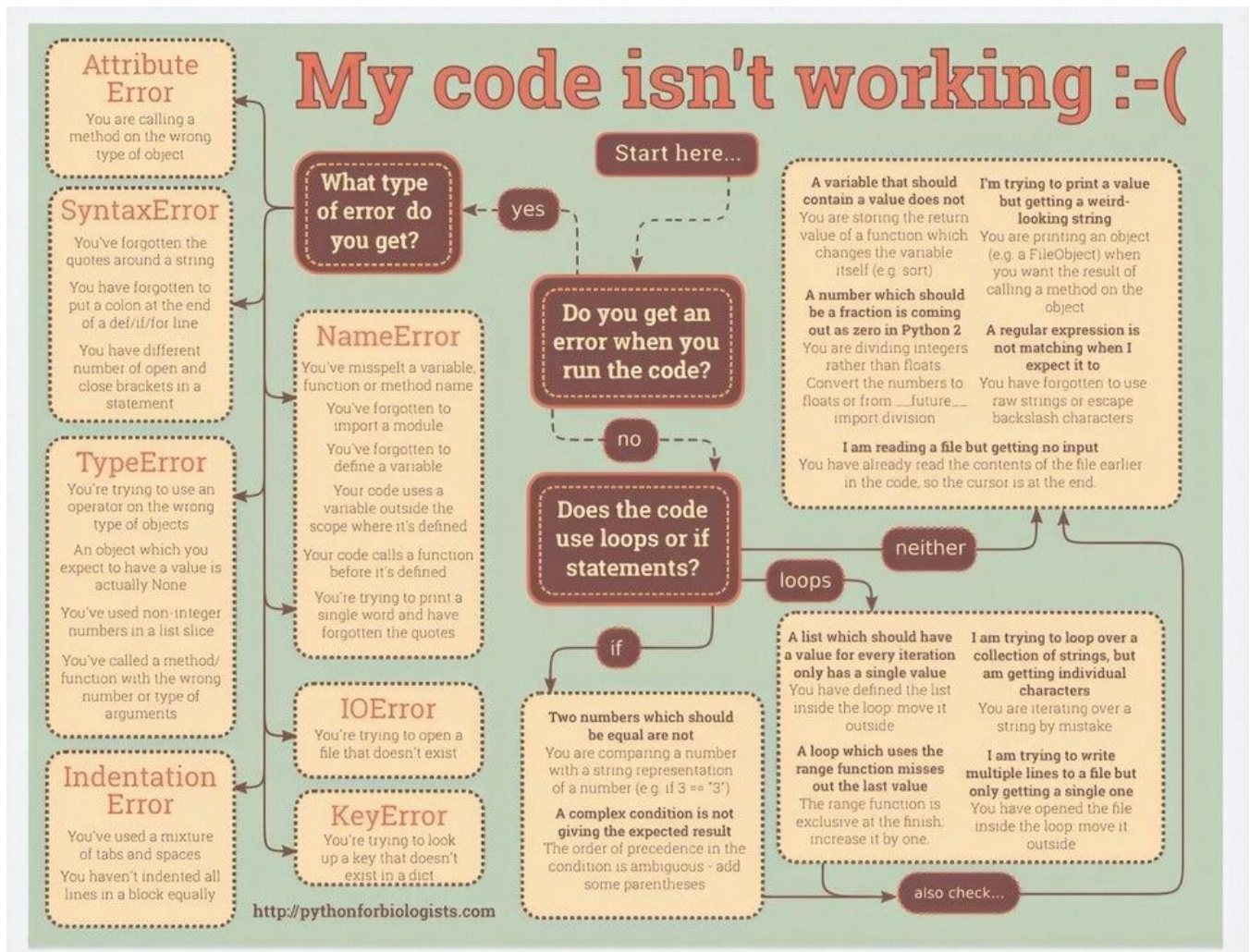
If you want to learn to program, plan to spend a lot of time in front of your computer, not just reading, but programming as well.

Neural pathways are **strengthened** into habits through the repetition and practice of thinking, feeling and acting.

PRACTICE: Start your morning passionately declaring aloud your goals for the day.

Declarations send the power of your subconscious mind on a mission to find solutions to fulfill your goals.

My Code Isn't Working 😞



Glossary

assignment A statement that assigns a value to a variable.

concatenate To join two operands end to end. Typically used with strings and string literals.

comment Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

evaluate To simplify an expression by performing the operations in order to yield a single value.

expression A combination of variables, operators, and values that represents a single result value.

floating point A type that represents numbers with fractional parts.

integer A type that represents whole numbers.

keyword A reserved word that is used by the compiler to parse a program; you cannot use keywords like if, def, and while as variable names.

mnemonic A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

modulus operator An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.

operand One of the values on which an operator operates.

operator A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

statement A section of code that represents a command or action. So far, the statements we have seen are assignments and print expression statement.

string A type that represents sequences of characters.

type A category of values. The types we have seen so far are integers (type int), floating-point numbers (type float), and strings (type str).

value One of the basic units of data, like a number or string, that a program manipulates.

variable A name that refers to a value.

Assignment Submission

Attach all program files to the assignment in BlackBoard.